Technieken voor het renderen van volumetrische objecten

D. Rebel (6v1) en F. Davidson (6v3)

Natuur en techniek, informatica

Metis Montessori Lyceum, Amsterdam

Begeleid door Monique Dewanchand

Ingeleverd op 20 Januari 2023

Inhoud

Inleiding

De meest gebruikte techniek van rendering in computer graphics is rasterising, waarbij een aantal driehoeken op het scherm worden getekend, en elke pixel een verschillende kleur kan worden toegewezen. Deze techniek werkt voor bijna alle modellen en effecten die gerendered worden in computer graphics. Maar voor sommige modellen of effecten kan deze techniek niet het gewenste resultaat produceren, of ten minste niet zonder dure tussenstappen. Een voorbeeld van een van deze effecten is bij het visualiseren van CT- of MRI-scans, waarbij meerdere dwarsdoorsnedes van een object, meestal van een gedeelte van een lichaam, de input data is. Andere voorbeelden zijn fotorealistische wolken of rook. Deze data is niet op een duidelijke manier te transformeren tot de driehoeken die als input data nodig zijn voor het rasterising algoritme. In dit verslag vergelijken we drie alternatieve rendering technieken om dit soort 'volumetrische' data weer te geven.

We zullen de drie verschillende algoritmes implementeren, en runnen op verschillende hardware, om de snelheid van de verschillende algoritmes te vergelijken, en om te kijken hoe dat verschilt op verschillende machines. Voor alle implementaties gaan we openGL gebruiken, om de berekeningskracht van de GPU te gebruiken, zoals dat vaak gedaan wordt in grafische programma's. Omdat deze technieken vaak gebruikt worden in de medische beeldverwerking en analyse, zullen wij de algoritmes laten runnen op de CT-scan data van een schedel. De CT-scan data is verkregen van AMRG Cardiac Atlas Project (http://www.cardiacatlas.org/studies/amrg-cardiac-atlas/). Naast de snelheid van de algoritmes zullen we in dit verslag ook de output van de algoritmes vergelijken op hoe realistisch ze eruitzien, en hoe goed details in de data te zien zijn.

De algoritmes die wij hebben gekozen om te vergelijken zijn: het marching cubes algoritme, direct volume rendering en texture slicing. Deze technieken hebben wij gekozen omdat ze erg verschillen in aanpak van het probleem. Hierdoor kunnen we een betrouwbare conclusie trekken over wat voor algoritme het beste werkt voor verschillende doelen. Bij een Disneyfilm ligt de nadruk niet op snelheid, maar wel op hoe fotorealistisch het resultaat eruitziet. Daarentegen is bij real-time graphics een nadruk op snelheid, en is het niet erg om een minder realistisch beeld te hebben. Waar de doelen verschillen, verschillen vaak ook de middelen.

Toepassingen van de technieken

Met de technieken die wij behandelen kunnen driedimensionale datasets gevisualiseerd worden. Driedimensionale datasets komen vaak voor bij CT- en MRI-scans, vloeistofsimulaties of aardbewegingsmetingen. Deze technieken zijn van toepassing bij alle driedimensionale datasets waar oppervlaktes niet duidelijk te onderscheiden zijn.

Deze datasets bestaan uit lange reeksen van getallen. Als deskundigen deze datasets zo moesten gebruiken zouden ze niet een goed beeld krijgen van wat de data voorstelt. Bijvoorbeeld moeten doctors een goed beeld hebben van de anatomie van de patiënt zodat er een probleem kan worden vastgesteld en een juiste diagnose kan worden gegeven. Door een beter beeld te hebben van de anatomie van de patiënt kunnen de doctoren dus ook een beter oordeel doen over een toepasselijke diagnose. Ontwikkelingen in het visualiseren van de datasets die gegenereerd worden door een scan kunnen er meer details onderscheiden worden.

Door scans worden ook minder invasieve operaties ondergaan, wat voor de patiënt minder ongemak en onrust, geen hersteltijd en het voorkomen van een dure ziekenhuisovernachting kan betekenen. (Medical Imaging of Fredericksburg [MIF], z.d.) Deze scans duren ook minder lang en zijn vaker uit te voeren dan een operatie, en zo kan er dus ook tot een eerdere diagnose worden gekomen. Bij ziektes die snel ontwikkelen, of moeilijk op te merken zijn van een buitenperspectief is dit van belang, zodat de ziekte in een vroeg stadion kan worden gedetecteerd en genezen voordat de ziekte tot een levensbedreigende fase komt.

Naast toepassingen in de medische industrie zijn er ook veel toepassingen in de olie-industrie, met betrekking tot de seismische tomografie. Bij seismiek worden seismische golven op de grond gezonden. De seismische golven zijn golven die zich door de aarde voortplanten, en kunnen opgewekt worden door mensen maar komen ook vrij bij aardbevingen. De lagen in de aarde weerkaatsen die golven, en deze golven worden weer ontvangen. Met de golven die op verschillende plekken worden opgevangen kan een driedimensionale dataset van de materialen die zich in de grond bevinden worden gemaakt. Deze dataset kan dan worden gevisualiseerd en geanalyseerd om details over de ondergrond te weten te komen zonder de grond aan te tasten (Chaves et al., 2011).

De visualisatie van driedimensionale datasets van seismisch onderzoek is erg belangrijk voor het detecteren van afwijkingen, begrijpen van ondergrondse reservoirs, en het herkennen van facies. Het herkennen van welke facies er in de grond zitten kan geologen ondersteunen met het herkennen van natuurlijke processen in de omgeving. Het begrijpen van ondergrondse reservoirs is belangrijke informatie voor het vinden van nieuwe brandstof zoals aardolie. Het detecteren van afwijkingen geeft informatie over hoe veilig de ondergrond is om op te bouwen. Door de structuur te analyseren kunnen er ook patronen herkend worden die verbonden worden met bepaalde mineralen, of kunnen archeologen plaatsen van archeologisch interesse vinden. Zo is binnen de seismische en medische tomografie veel gebruik voor het gedetailleerd visualiseren van driedimensionale datasets.

Maar ook buiten de seismische en medische tomografie worden deze technieken gebruikt. In de industrie van ‘Computer Generated Imagery’ (CGI) zijn deze technieken ook vaak van toepassing. Onder CGI vallen onder andere video games en films met (gedeeltelijk) digitale beelden. Als er met CGI een

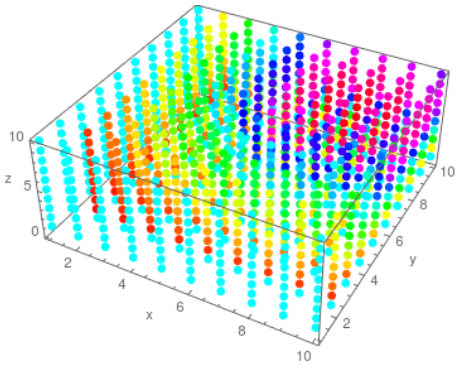
beeld dat lijkt op realistische mist, rook, wolken, troebel water, of iets dergelijks moet worden gegenereerd, wordt dat gedaan met volumetrisch renderen.

De industrie van CGI is de laatste decennia veel gegroeid, en een groot deel van de films die nu uitgebracht worden gebruiken CGI. Bij films is het doel vaak om de gegenereerde beelden zo fotorealistisch als mogelijk te maken, om de kijker niets te laten merken van dat de beelden gegenereerd zijn door een computer. Dit is mogelijk voor films, omdat de films eerste gerendered worden, de beelden worden opgeslagen en dan laten zien. In games is dit anders, omdat een game moet reageren op wat de gebruiker doet, moet het beeld gerendered worden op het moment dat de gebruiker de game speelt. Het is binnen de industrie standaard om elke 16.67 milliseconden een nieuw beeld te laten zien, wat betekent 60 ‘frames’ elke seconde. Soms, voor erg grafisch intensieve games, word gekozen voor 30 of 24 frames per seconde. Als een game op 60 frames per seconde runt, dan moet het beeld binnen 16.67 milliseconden gerendered worden. Bij games maakt de snelheid van de algoritmes heel erg uit, omdat de gebruiker anders op kan merken dat er geen vloeiend beeld is, maar een reeks stille beelden. In films worden dus vaak langzamere en realistischere technieken gebruikt, en voor games vaak snellere, minder realistische technieken.

Wat is een volumetrisch object?

Een volumetrisch object is moeilijk te omschrijven op een traditionele manier, met driehoeken. Bij traditioneel rasterisen wordt een oneindig dunne schil om het object heen weergegeven. Deze schil wordt gemaakt van veel (soms honderdduizenden) driehoeken. Bij volume rendering technieken wordt het hele object, inclusief binnenkant, meegenomen bij de berekeningen die bepalen welke kleur elke pixel op het scherm moeten worden. Voor een ander eindresultaat moet het traditionele rasterising algoritme worden aangepast of vervangen door een nieuw algoritme. Deze algoritmes behandelen we later in dit verslag.

Omdat er andere berekeningen worden gedaan met volumetrische rendering technieken, is er ook een andere datastructuur vereist. Er zijn verschillende manieren om een driedimensionaal scalair veld in een computer op te slaan, en wij hebben gekozen voor een driedimensionaal rooster. Hierin worden op een arbitraire resolutie scalaire waarden opgeslagen.

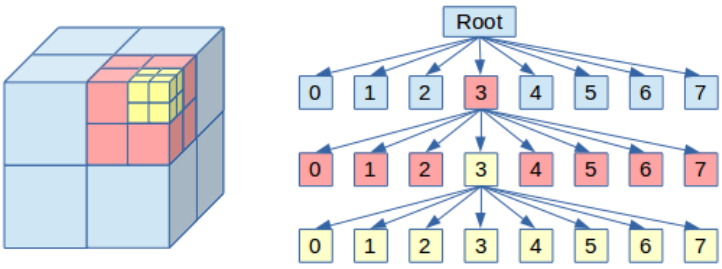


Als er data uit deze driedimensionale ‘texture’ moet worden gehaald, moet je aangeven van waar deze waarde moet komen in drie dimensies. De simpelste manier om deze waarde uit de dataset te halen is om te berekenen welke texel (texture pixel) het dichtst bij de gevraagde locatie ligt, en dan die waarde te gebruiken. Dit heet nearest-neighbour texture filtering. Deze manier heeft wel een probleem, als je met een lage resolutie werkt of erg dicht op de texture zit, dan zijn er duidelijke grenzen tussen de texels te zien. Dit kan een minder realistisch beeld veroorzaken. Als alternatief tot nearest-neighbour texture filtering is er bilinear texture filtering. Bij bilinear texture filtering wordt er gekeken naar de acht dichtstbijzijnde texels (acht voor drie dimensies, vier voor twee dimensies, twee voor één dimensie), en wordt er afhankelijk van de samplelocatie een mix tussen die texels gebruikt. Meestal zijn textures tweedimensionaal, dus vandaar de naam bilinear texture filtering. Wij gebruiken voor driedimensionale datasets driedimensionale textures.

Als de texture van veraf wordt bekeken, zou de texture zo klein kunnen worden dat het in een enkele pixel valt. Als er dan maar naar acht texels wordt gekeken, zal dit niet een goed resultaat leveren, omdat er maar naar een klein deel van de texture wordt gekeken. Vaak kan dit probleem tot een ongewenst

glinsterend effect leiden. Er is nog een andere techniek die wordt gebruikt om textures te samplen, mipmapping. Een mipmap is een vooraf berekende reeks textures die de vorige texture vertegenwoordigt op lagere resolutie. De resolutie wordt elke iteratie gehalveerd. Elke texture met een lagere resolutie is dan het volgende mipmap niveau. Dit gaat door tot er een texture van 2 bij 2 bij 2 texels is. Afhankelijk van de afstand tot de texture worden twee mipmap niveaus uit deze reeks gekozen die het meest gepast zijn, en wordt bilinear texture filtering toegepast op beide textures. De uiteindelijke waarde is dan een mix van beide hiervoor verkregen waardes. Deze techniek heet trilinear texture filtering, omdat er lineair geïnterpoleerd wordt tussen twee waarden die met bilinear texture filtering zijn verkregen van verschillende mipmap niveaus.

Een andere manier om een driedimensionaal scalair veld op te slaan is met een ‘octree’. Een octree is een recursieve datastructuur, en het lijkt op een driedimensionaal rooster, behalve dat de resolutie niet op elke plek hetzelfde is. Op plekken waar weinig detail zit, en alle waarden dicht bij elkaar zitten, kan een lagere resolutie gekozen worden zonder veel detail te missen, en op plekken waar veel detail zit, en er veel onderling verschil zit in nabije waarden, kan een hogere resolutie gekozen worden. Een octree is een verzameling aan nodes, en elk van de nodes kan acht sub-nodes hebben, of niet. Deze sub-nodes kunnen dan ook weer acht sub-nodes per stuk hebben, enzovoort. Elke node die niet sub-nodes heeft, slaat een waarde op.



Als er een waarde uit de octree moet worden opgezocht, dan wordt er bij de huidige node gekeken of deze een waarde heeft of dat deze sub-nodes heeft. Als het een waarde heeft, dan wordt de waarde gegeven. Zo niet, dan wordt gekeken welke sub-node het dichtst bij de gevraagde samplelocatie, en wordt voor die node gekeken of die sub-nodes heeft. Dit gaat door totdat er een node is gevonden met een waarde. Die waarde is dan de waarde op de samplelocatie.

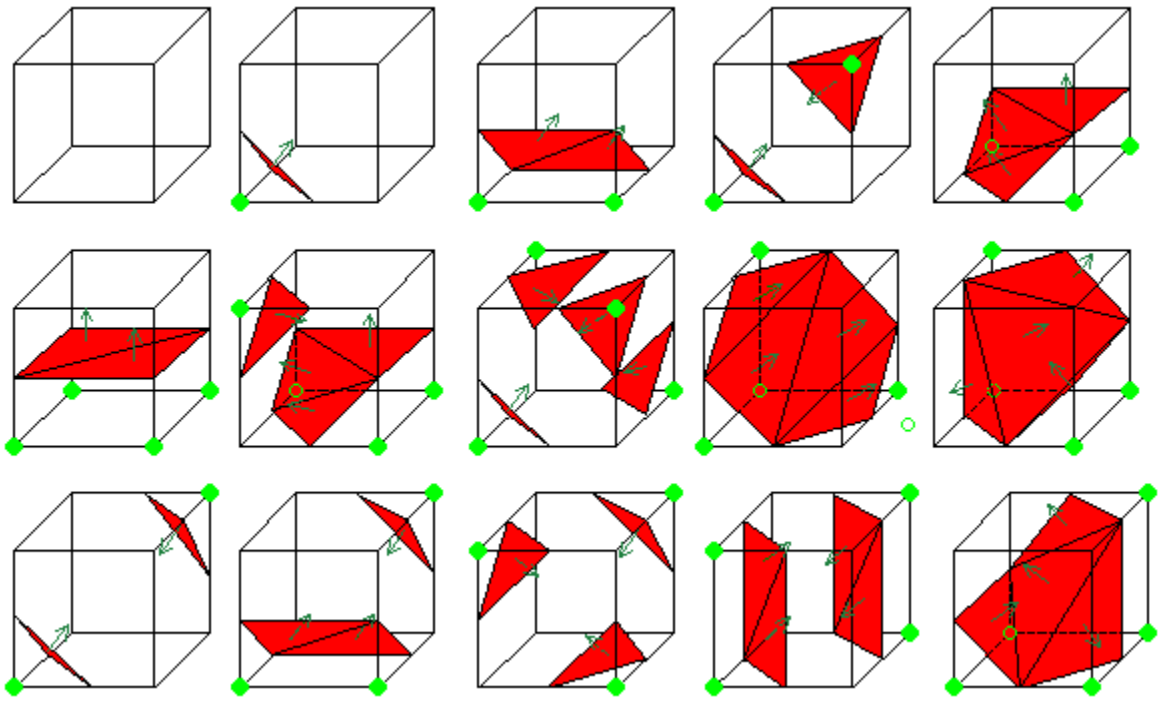
Omdat er op plekken van laag detail minder nodes zitten, wordt hier ook geheugen van de computer bespaard. Voor datasets die plekken met laag detail hebben en plekken met hoog detail hebben kan deze datastructuur minder geheugen innemen dan een driedimensionale texture met één constante resolutie. Een nadeel van deze datastructuur is dat voor datasets met een relatief constant niveau aan detail het meer geheugen kan innemen dan een texture. Het monsteren van de data duurt ook langer dan bij een texture. Voor onze applicaties zullen wij een rooster gebruiken in plaats van een octree.

Techniek 1: Marching cubes

De techniek maakt de scalaire waarden van het veld binair door te kijken of de waarden groter zijn dan een gekozen waarde. Als het hoger is dan de gekozen waarde, is het zichtbaar en als het kleiner is dan de gekozen waarde is het onzichtbaar. Door de waarde te veranderen kan je in medische beeldverwerking verschillende soorten weefsel met elkaar onderscheiden, wat het beeld veel overzichtelijker kan maken.

Het marching cubes algoritme trekt als het ware een grens tussen onzichtbare en zichtbare waardes. De grenzen tussen de zichtbare en onzichtbare waarden worden omschreven door driehoeken, die dan gerasterised worden. Deze grenzen hebben wij gemaakt door een look-up-table van http://paulbourke.net/geometry/polygonise . In deze look-up-table staan de oplossingen voor alle mogelijke binaire waardes voor de acht hoekpunten van een kubus. In deze oplossingen staan welke driehoeken er moeten zijn om de grens tussen alle waarden te leggen. Zo kan je voor een rooster een grens leggen tussen de verschillende binaire waarden. In het figuur hieronder is te zien hoe deze grenzen eruitzien.

De driehoeken die dan zijn gegenereerd worden dan gerasterised. Maar vaak is het handig om de gekozen waarde tussen zichtbaar en onzichtbaar te kunnen verstellen, om zo te zien hoe de grenzen veranderen voor verschillende waarden. Als deze waarde versteld wordt, is het model incorrect en moet het hele model opnieuw worden gegenereerd.



Techniek 2: Direct volume rendering

In tegenstelling tot het marching cubes algoritme, berust direct volume rendering niet op het traditionele rasterising algoritme. In plaats van rasterising wordt bij direct volume rendering ray tracing gebruikt. Bij ray tracing wordt voor elke pixel een lijn opgesteld, ook wel een 'ray' genoemd, vandaar de naam ray tracing. Voor elk object wordt gecheckt of deze lijn door het object heen gaat, en de pixel krijgt de kleur van het object dat het dichtst bij geraakt wordt door de lijn.

Voor elke pixel die door een volumetrische kubus gaat, moet de doorlaatbaarheid worden berekend. De kleur achter de volumetrische kubus moet ook berekend worden, want als de volumetrische kubus niet al het licht stopt, komt het licht van achter de kubus ook in de camera, en moet dus berekend worden. Er zijn verschillende manieren om de kleur en doorlaatbaarheid van een volumetrisch object te berekenen. Een aantal bekende en accurate worden omschreven in een artikel van Ljung et al. (2016) Om de doorlaatbaarheid van een rechte lijn door de volumetrische kubus te berekenen zijn meerdere variabelen nodig, de in- en uitgangspositie van de lijn en de driedimensionale texture van de dichtheid voor elk punt in de kubus.

(1) (Max, 1995)

$$\frac{I}{I_0} = \left(\frac{1}{2}\right)^{\varepsilon \int_{p_{in}}^{p_{uit}} \left(d(\vec{p})\right) d|\overrightarrow{p_{uit}} - \overrightarrow{p_{in}}|}$$

Waar:

I de waargenomen intensiteit is,

I_0 de uitgezonden intensiteit van de lichtbron is,

ε een constante afhankelijk van de stof is,

P_{in} de ingangspositie van de lijn is,

P_{uit} de uitgangspositie van de lijn is,

$d(\vec{p})$ is de dichtheid van het volume op locatie $\vec{p}$

Maar er is een probleem met deze functie. Een computer kan deze functie niet oplossen, omdat we niet de dichtheidsfunctie kunnen primitiveren, omdat er arbitraire data in opgeslagen is. Om dit op te lossen wordt de integraal benaderd door de Riemann-som van de functie te nemen. Door de resulterende functie anders op te schrijven krijg je de onderstaande functie.

(2) (Max, 1995)

$$\frac{I}{I_0} = \prod_{i=1}^{n} \left(\frac{1}{2}\right)^{d(\overrightarrow{p_i}) \frac{|\overrightarrow{p_{uit}} - \overrightarrow{p_{in}}|}{n}}$$

Deze functie komt overeen met de wet van Beer-Lambert, die luidt:

(3)

$$\log \frac{I}{I_0} = -\varepsilon\,[A]l$$

Wat herschreven kan worden als:

$$\log_2 \frac{I}{I_0} = -\varepsilon\,[A]l\ \log_2(10)$$

Waar:

I de waargenomen intensiteit is,

I_0 de uitgezonden intensiteit van de lichtbron is,

ε een constante afhankelijk van de stof is,

$[A]$ de gemiddelde concentratie van de stof in het volume is,

l de afgelegde afstand door de stof is.

De wet van Beer-Lambert kan herschreven worden als:

(4)

$$\frac{I}{I_0} = \left(\frac{1}{2}\right)^{\varepsilon[A]l\ \log_2(10)}$$

Formule 4 en 1 kunnen gelijk aan elkaar gesteld worden, waarna duidelijk te zien is dat

$$[A]l \sim \int_{P_{in}}^{P_{uit}} (d(\vec{p}))\ d|\overrightarrow{p_{uit}} - \overrightarrow{p_{in}}|$$

Het is intuïtief af te leiden dat de afstand tussen het in- en uitgangspunt de afgelegde afstand door de stof is. Ook is af te leiden dat het integraal van de dichtheidsfunctie tussen het in- en uitgangspunt gedeeld door de afgelegde afstand door de stof gelijk is aan de gemiddelde dichtheid. De vergelijking klopt dus, en omdat formule 1 en 2 gelijk zijn als de waarde van n oneindig is, zal formule 2 betrouwbare resultaten geven als een hoge waarde voor n wordt gekozen. Een computer kan niet een oneindig aantal berekeningen doen, dus n kan niet een waarde van oneindig hebben. Hoe hoger de waarde van n, hoe meer detail in de dichtheidsfunctie zal worden meegerekend, hoe langer de computer zal doen over de berekeningen. De gebruiker kan de waarde van n veranderen om een goed balans tussen kwaliteit en snelheid te vinden.

Bij deze formules voor de doorlaatbaarheid wordt berekend hoeveel licht er de camera binnen komen, zonder afgebogen te worden door de stof. Maar in de echte wereld kan er ook licht dat niet naar de camera toe gaat afbuigen naar de camera. Om dit te berekenen moet voor elk punt $\overrightarrow{p_i}$ berekenen hoeveel licht van een lichtbron het redt naar het punt, welk percentage van het licht afgebogen wordt in de richting van de camera op dat punt, en welk percentage van dat afgebogen licht het redt naar de camera.

$$\frac{I}{I_0} = \sum_{i=1}^{n} \frac{I}{I_{p_i}} \cdot a \cdot \frac{I_{p_i}}{I_0}$$

Waar:

I de waargenomen intensiteit is,

I_0 de uitgezonden intensiteit van de lichtbron is,

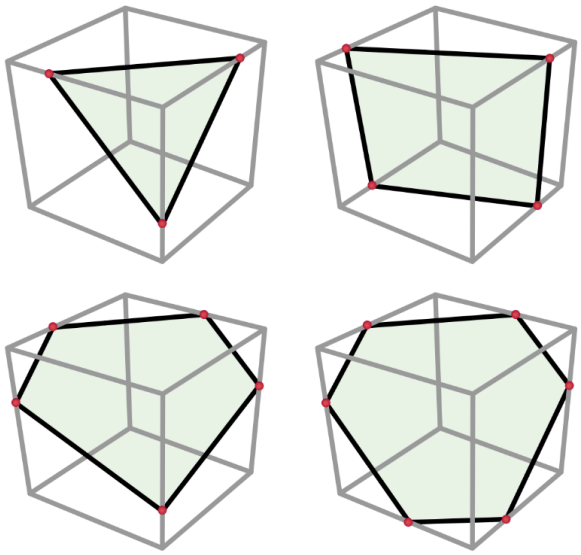
I_p de intensiteit die punt $\overrightarrow{p_i}$ bereikt is,

a het percentage van het licht dat afgebogen wordt in de richting van de camera op punt $\overrightarrow{p_i}$

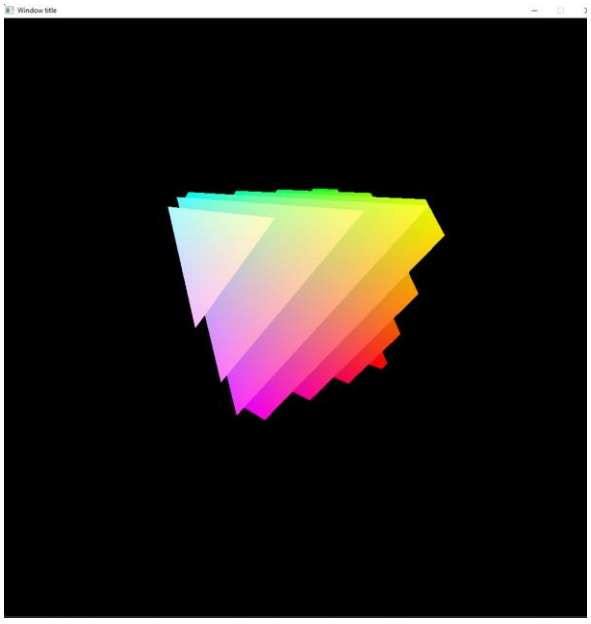
Met deze formule kan je ook nog de kleur van het volumetrisch object berekenen. De uiteindelijke kleur van de pixel wordt voor een gedeelte de kleur van het volumetrisch object en een gedeelte de kleur van het object achter het volumetrisch object.

Techniek 3: Texture slicing

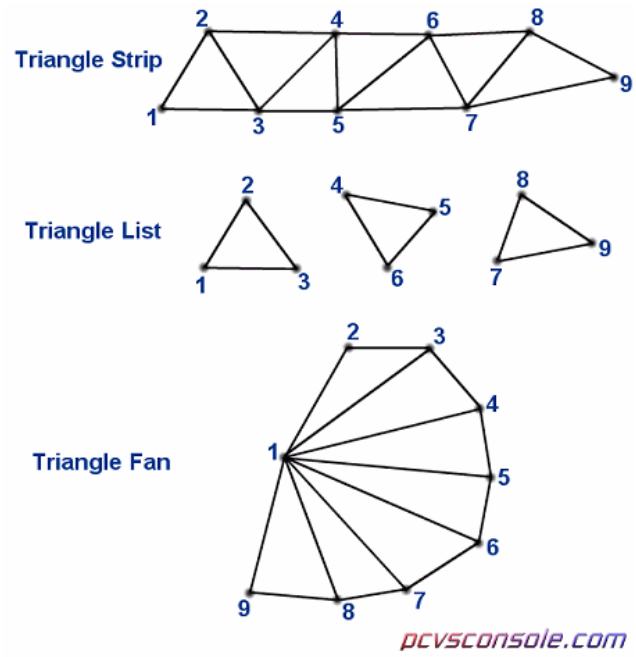
Een andere techniek om een driedimensionale texture te visualiseren is door als het ware plakjes van de texture te rasterizen. Op deze plakjes wordt dan een 2-dimensionaal plaatje geplakt, dat een intersectie van het 3D object dat gerendered wordt vertegenwoordigt. Deze plakjes worden dan achter elkaar gerenderd, waardoor er diepte ontstaat. De wiskunde die achter deze intersectiepunten berekenen zit, wordt uitgelegd op cococubed.com. (Volume of Intersection Between a Plane and a Cube, n.d.) Omdat de plakjes altijd precies loodrecht op het zicht van de camera moeten zijn, moeten deze plakjes elke keer dat de camera verplaatst opnieuw worden gegenereerd.



Deze plakjes worden gegenereerd door de snijpunten van een vlak en de kubus van het volume te berekenen. Een kubus heeft 8 hoekpunten. Deze worden verbonden met 12 lijnen. Voor elke lijn wordt berekend of en zo ja, wáár het vlak deze lijn zal snijden. Nu we alle punten weten die op de hoeken van de plak zitten, kunnen we het nog net niet renderen. We moeten eerst nog weten op welke volgorde deze punten zijn, zodat de correcte driehoeken worden opgesteld.



Omdat vaak driehoeken aan elkaar liggen, zijn er vaak gedupliceerde hoekpunten van de driehoeken in het geheugen, als voor elke driehoek elk hoekpunt achter elkaar in het geheugen moet staan. Om dit op te lossen zijn er verschillende primitieven in openGL. Enkele voorbeelden zijn triangle strip, triangles (triangle list genoemd in het plaatje), en triangle fan. Het simpelste primitief is het triangles primitief. Er wordt een driehoek gemaakt van hoekpunt 1, 2, en 3. De volgende wordt gemaakt van hoekpunt 4, 5, en 6. Enzovoort. Maar voor sommige situaties is dit niet het meest efficiënt gebruik van geheugen. Om meer efficient bijvoorbeeld lijnen met gevarieerde dikte te tekenen is de primitief triangle strip handig, die een driehoek maakt met hoekpunt 1, 2 en 3, en de volgende maakt met 2, 3, en 4. Enzovoort. Voor convexe polygonen is de primitieve triangle fan handig, die een driehoek maakt met hoekpunt 1, 2, en 3, en de volgende maakt met 1, 3, en 4, de volgende met 1, 4, en 5. Enzovoort.



Van de hoekpunten die we te weten krijgen uit de vlak-kubus intersecties weten we dat ze een convexe polygoon kunnen vormen, en dat is de vorm die we willen. Wij gebruiken dus de primitieve triangle fan. Dan moeten de hoekpunten nog in de goede volgorde om correct gerendered te worden. Omdat we het primitieve triangle fan gebruiken en we weten dat we met een convexe polygoon werken, weten we dat de juiste volgorde de volgorde is die de kortste route langs alle hoekpunten geeft.

Om de kortste route te berekenen van een hoekpunt, langs alle hoekpunten en weer terug naar het beginpunt gebruiken wij het travelling salesman algoritme. Een bezorgdheidpunt bij deze keuze is de tijdscomplexiteit van dit algoritme. Dit algoritme heeft een tijdscomplexiteit van O(n!) of O(n^2 * 2^n), afhankelijk van implementatie, waar n het aantal punten is. Dit is niet goed, omdat het algoritme dan erg snel langzamer wordt met het aantal punten. Maar voor ons gebruik is dit niet een groot probleem, omdat in onze situatie n nooit groter dan 6 wordt, omdat de vlak-kubus intersectie nooit meer dan 6

hoekpunten geeft. Uit het algoritme krijg je ook de route die het snelste is, en op deze volgorde slaan wij de punten op, zodat de vorm altijd goed en compleet gerendered wordt.

Om een realistisch beeld te geven van de dataset moet er ook transparantie in de techniek verwerkt zitten, omdat niet al het licht in de echte wereld tegengehouden zal worden door het object. Het percentage van het licht dat het haalt door een volume met een bepaalde dichtheid kan worden berekend volgens de Beer-Lambertwet.

In een rasteriser kan je ook transparantie implementeren, alhoewel het moeilijk is in een algemene manier.

Als we een transparant ding willen renderen, en we weten hoeveel procent van het licht van het voorste (transparante) object komt, en we weten welke kleuren het voorste (transparante) object en de achtergrond hebben, dan kunnen we een uiteindelijke kleur berekenen. Deze kleur wordt berekend met de volgende formule:

$$C = C_{voor} \cdot \alpha + C_{achter} \cdot (1 - \alpha)$$

Waar:

C de uiteindelijke kleur is,

C_{voor} de kleur van het transparante object is,

C_{achter} de kleur van de achtergrond is en

α het percentage van licht dat van het transparante object komt is.

In computer graphics wordt dit ook wel ‘color blending’ genoemd. Voor deze formule is het wel belangrijk dat het transparante object vóór het object in de achtergrond wordt getekend, omdat anders de pixels van het object in de achtergrond worden geculled, omdat er een kleinere waarde in de depth buffer stond, terwijl het eigenlijk in de achtergrond had moeten verschijnen. Als dit fout gaat, dan verdwijnt een object achter een transparant object, en is door het transparante object verder in de achtergrond te zien dan hoort.

Daarom is het erg belangrijk dat in een scene met transparante objecten alles van achter naar voren wordt getekend. Dus eerst de objecten in de achtergrond, en dan de objecten voorin. Zo weet je zeker tijdens de berekening van transparante objecten dat de achtergrond niet gaat veranderen, zodat je niet een ongewenst effect hebt. Dit sorteren is vaak moeilijk, duur, of zelfs onmogelijk als twee concave transparante objecten in elkaar zitten. Bij concave objecten is deze techniek niet altijd kloppend omdat het licht eerst door object 1, dan door object 2, dan weer door object 1, en dan mogelijk weer door

object 2 zou kunnen gaan. Daarom is het moeilijk om een algemene techniek voor transparantie in een rasteriser te hebben.

Maar in onze situatie kunnen we bij het genereren van de plakjes ze allemaal achter elkaar zetten, en we weten dat ze niet concaaf zijn, dus het sorteren is voor ons geen probleem. Dan geven we alle plakjes dezelfde α-waarde. Door de manier waarop we de kleuren blenden, zal er een logaritmisch verband ontstaan tussen het aantal plakjes die op een pixel zitten en de invloed van de kleur. Dit is ook zo in de Beer-Lambertwet, omdat daar de lengte l ook in de macht staat.

Zo is deze techniek ook gedeeltelijk geaard in de Beer-Lambertwet, en kan het met genoeg plakjes een realistisch beeld geven van de dataset.

Resultaten

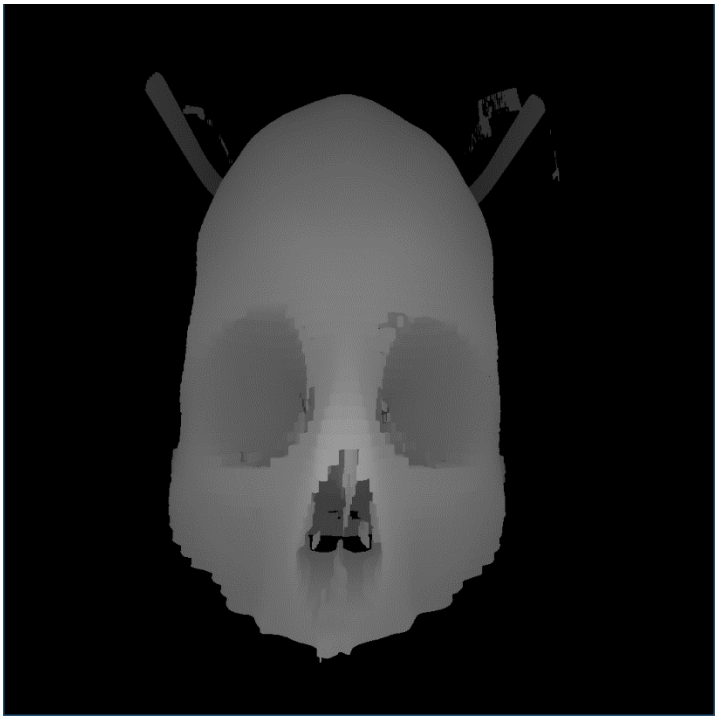
Deze resultaten zijn verkregen door op verschillende computers, met verschillende hardwarecomponenten en verschillende operating systems en drivers dezelfde tests te runnen. Deze tests zijn als volgt: degene die de test afneemt “vliegt” rondom en door het gerenderde schedel, en het programma slaat voor elke frame de tijd dat het duurde om die frame te generaten. Deze data is dan op verschillende manieren verwerkt. De eerste manier is het gemiddelde nemen van alle frame times, om de gemiddelde frame time en framerate te berekenen. De tweede manier is een manier om de “waargenomen performance” te berekenen. Dit is gedaan omdat een frame langer op het scherm wordt laten zien als het een hogere frame time heeft, en ziet het er dus niet goed uit als de gemiddelde framerate wel hoog is, maar er grote spikes in frame time zijn. Om deze weighted avg. te berekenen is de frame time zelf als gewicht bij het berekenen van het gemiddelde gebruikt.

Voor technieken waarbij iets aan het begin van het programma berekend moet worden dat een lange tijd duurt, is ook die tijd opgeslagen. Dit is het geval bij de marching cubes, waarbij de mesh gegenereerd moet worden.

marching cubes	computer 1 (amd mesa)	computer 2 (amd windows)	computer 2 (amd mesa)	computer 3 (nvidia windows)	computer 3 (intel windows)
mesh gen time (s)	1.8147275	2.67314	3.743014	2.34326	2.398056
avg. frame time (ms)	0.015755628	0.145971202	0.053733383	2.160078947	4.845526316
avg. framerate (fps)	63469.38561	6850.66636	18610.40453	462.9460424	206.37593
weighted avg. frame time (ms)	0.033593734	0.255337573	0.067032249	2.341969348	6.212442446
weighted avg. framerate (fps)	29767.45594	3916.384064	14918.19254	426.991071	160.967286
raytracer					
avg. frame time (ms)	8.583819857	9.220116387	37.12684358	4.845526316	21.44329082
avg. framerate (fps)	116.498251	108.4585007	26.93468939	206.37593	46.63463311
weighted avg. frame time (ms)	30.63433807	22.8883093	45.80399058	6.212442446	32.7983451
weighted avg. framerate (fps)	32.6431078	43.69042671	21.83215889	160.967286	30.48934319
texture slicer					
avg. frame time (ms)	1.144456046	3.370136113	1.492034938	1.984021739	4.623183333
avg. framerate (fps)	873.7775503	296.7239205	670.2255923	504.0267353	216.3011778
weighted avg. frame time (ms)	2.272478241	4.673019824	3.521560919	2.086278796	5.014286017
weighted avg. framerate (fps)	440.0482177	213.9943843	283.9649869	479.3223236	199.4301874

Vergelijking van technieken

Marching cubes



Bij het marching cubes algoritme blijft de performance best wel hoog bij hoge resolutie. Hierdoor kan het gemakkelijk op grote schermen van dichtbij bekeken worden zonder dat het er wazig uit zal zien. Dit is handig voor het analyseren van medische scans.

De techniek gebruikt ook de traditionele rasterising techniek, waardoor het makkelijk te integreren en delen is als driedimensionaal model. Omdat het geen transparante eigenschappen heeft kan het ook gemakkelijk gerenderd worden in een scene met andere objecten ertussen.

Omdat het genereren van de mesh gedaan wordt op basis van een grenswaarde, is het ook makkelijk om de verschillende lagen van het lichaam wel en niet te laten zien. In het plaatje hierboven wordt bijvoorbeeld alleen nog maar het bot laten zien, maar het is ook mogelijk om ook het kraakbeen en vel te laten zien.

Echter kunnen medische scans van zo'n hoge resolutie zijn dat het genereren van een mesh een erg lange tijd kan duren. Om dit te voorkomen kan de scan effectief gedownscaled worden, waardoor er detail verloren gaat, maar heb je sneller een beeld. Anders zou het regenereren van de mesh te lang duren voor een goede en effectieve gebruikerservaring.

Omdat het niet doorzichtig wordt laten zien, kunnen er details verborgen zitten. Er kunnen ook objecten zijn die compleet omringd zijn door andere objecten, waardoor je van buitenaf niet weet dat het er is. Dit betekent dus dat je soms niet helemaal een goed beeld kan krijgen van wat je wilt zien.

Direct volume rendering

In theorie is direct volume rendering de meest realistische techniek. De stappen die in het algoritme doornomen worden zijn namelijk dezelfde als die in de echte wereld. Hierdoor zal het ook met meest realistisch resultaat geven. Je kan dus, in tegenstelling tot het marching cubes algoritme een detail in het midden van de dataset zien, ook al zit er nog iets voor.

Een nadeel aan deze techniek is wel dat het erg slecht schaalt bij hoge resolutie en al helemaal niet snel runt op oude en/of low-end GPU's. Er zijn een aantal waarden die aangepast kunnen worden om hogere performance te bereiken, in ruil voor minder realistisme. Bij het opnemen van de performance van de verschillende technieken hebben we 40 sampling stappen en 8 in-scattering stappen gedaan.

Dat deze techniek niet zo goed runt op oude en/of low-end GPU's is omdat die GPU's er nog vaak niet voor gemaakt zijn om te raytracen, maar om te rasterisen. De hele GPU-pipeline is geoptimaliseerd voor het rasterisen, en met een raytracer kan hier niet goed gebruik van worden gemaakt omdat de twee technieken fundamenteel anders in werking gaan. De GPU-fabrikanten zijn ook bezig met een ray

tracing pipeline die de GPU meer efficiënt aanbiedt aan de programmeur voor ray tracing rendering technieken.

Texture slicing

De snelheid van de texture slicing techniek is hoog voor de kwaliteit van het beeld, dit komt omdat het efficiënt gebruik kan maken van de rendering pipeline die geoptimaliseerd is voor rasterisen. Deze techniek heeft ook niet veel van de nadelen van marching cubes, omdat je hierbij wel transparantie hebt in je object. In deze zin is het een realistischer model dan het marching cubes algoritme. Een nadeel relatief tot marching cubes is dat de dataset informatie niet opgeslagen is in de mesh, maar in de texture. Je kan de mesh dus niet gemakkelijk delen, omdat het een onorthodoxe texture mapping techniek gebruikt die niet wordt ondersteund door 3D-modelleersoftware zoals Blender. Je moet ook de mesh opnieuw berekenen als je van perspectief verandert.

Conclusie

In dit onderzoek hebben we verschillende manieren van het renderen van volumetrische objecten vergeleken, op basis van de visuele kwaliteit en de performance. Uit dit onderzoek is gebleken dat de verschillende technieken beter zijn voor verschillende doeleinden. Direct volume rendering kan gebruikt worden op plekken waar het handig is als het beeld fotorealistisch is, in tegenstelling tot het marching cubes algoritme.

Het marching cubes algoritme verandert de data in driehoeken die makkelijk gerenderd kan worden door de GPU. Er wordt hierbij een grenswaarde genomen tussen wat vast is en wat hard is, waardoor er afhankelijk van waar je naar wilt kijken verschillende dingen kunnen worden laten zien. Deze is heel snel nadat de mesh is gegenereerd, maar het genereren zelf kan heel sloom zijn. De techniek ruilt realisme in voor snelheid, door geen transparantie mee te nemen in de berekeningen, om de efficiënte rasterising pipeline van GPU's te benuttigen. Marching cubes is geschikt voor het visualiseren van seismische tomografische scans, en het wordt ook vaak gebruikt voor verschillende objecten zoals terrein in games.

Direct volume rendering gebruikt ray tracing om de kleur en felheid van elke pixel te berekenen. Dit is erg realistisch, omdat het de natuurkundige wetten van de echte wereld volgt. Deze techniek produceert ook het meest realistisch beeld van de drie technieken die wij behandelen. Dit realisme komt met een prijs, en dat is de snelheid van het algoritme. De snelheid van het algoritme is de laagste van de drie, vooral als het volumetrisch object van dichtbij bekeken wordt. Deze techniek is het best toepasbaar in situaties waar fotorealisme waardevol is, en er snelle computers beschikbaar zijn, of dat de berekening lang mag duren. Bijvoorbeeld CGI in films.

Texture slicing werkt door het volumetrisch object op te splitsen in plakken die loodrecht staan op de richting van de camera. Op deze plakken wordt dan een texture geplakt met transparantie zodat de plakken erachter ook zichtbaar zijn. Deze techniek is heel snel omdat de plakken erg makkelijk te representeren zijn als driehoeken, en produceert redelijk realistische resultaten. Omdat de plakken loodrecht op de camera moet blijven, moeten de plakken elke keer dat de camera bewogen wordt opnieuw berekend worden. Texture slicing gebruikt de efficiënte rasterising pipeline van GPU's met een model die gedeeltelijk gebaseerd is op de natuurkundige wetten van de echte wereld, waarmee het een balans tussen snelheid en realisme vindt.

Literatuurlijst

Chaves, M. U.,Di Marco, L.,Kawakami, G., & Oliver, F. (2011). Visualization of Geological Features Using Seismic Volume Rendering, RGB Blending and Geobody Extraction. 12th International Congress of the Brazilian Geophysical Society & EXPOGEF, Rio de Janeiro, Brazil, 15–18 August 2011,.

Ljung, P.,Krüger, J.,Groller, E.,Hadwiger, M.,Hansen, C. D., & Ynnerman, A. (2016). State of the Art in Transfer Functions for Direct Volume Rendering. Computer Graphics Forum,.35(3).

Max, N. (1995). Optical models for direct volume rendering. IEEE Transactions on Visualization and Computer Graphics,.1(2).

Medical Imaging of Fredericksburg. (z.d). How Medical Imaging Impacts Your Healthcare. Geraadpleegd op 14 January 2023, van https://mifimaging.com/2017/12/20/how-medical-imaging-impacts-your-healthcare/

Volume from a plane slicing a cube. (z.d). Geraadpleegd op 18 October 2022, van https://cococubed.com/code_pages/raybox.shtml

Bijlagen

https://github.com/Davidrbl/volumetric_raytracer_pws

https://github.com/Davidrbl/marching_cubes_ogl

https://github.com/Davidrbl/texture_slicer